# Parallel Computers and Complex Systems

*Geoffrey C. Fox*

**CRPC-TR92266**
**December 1992**

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

*From Biology to Computation, Inaugural Australian National Conference on Complex Systems*

SCCS - 370

# Parallel Computers and Complex Systems

by
Fox, Geoffrey C.

*Complex Systems '92: From Biology to Computation*
Innaugural Australian National Conference on Complex Systems

December 1992

Syracuse Center for Computational Science
Syracuse University
111 College Place
Syracuse, New York 13244-4100
<sccs@npac.syr.edu>
(315) 443-1723

# Parallel Computers and Complex Systems

Geoffrey C. Fox
Syracuse University
Northeast Parallel Architectures Center
111 College Place
Syracuse, New York 13244
gcf@nova.npac.syr.edu

September 30, 1992

**Abstract**

We consider parallel computing as the mapping of one complex system—typically a model of the world—into another complex system—the parallel computer. We study static, dynamic, spatial and temporal properties of both the complex systems and the map between them. The result is a better understanding of which computer architectures are good for which problems, software structure, automatic partitioning of data and performance of parallel machines.[1]

## 1 Introduction

Over the last dozen years, my group has been developing applications and the necessary software support for parallel computing [1]–[5]. During that time my work has made use of both physics and computer science ideas—the two areas I know something about. I have found that it has often been very helpful to view both the application, software and computer as systems, which we will call complex systems. Viewing these as physical systems, we introduce in Section 2 the concepts of space and time for complex systems. Section 3 describes spatial properties, size, topology, dimension and a physical analogy for data partitioning of adiabatic problems leading to concepts of temperature and phase transitions. In Section 4, we discuss temporal properties, a string model for very adaptive problems and a duality between the temporal structure of problems and the memory hierarchy of computers. In the final Section 5, we

[1]This work was supported by the Center for Research on Parallel Computation with National Science Foundation Coöperation Agreement No. CCR-9120008—the Government has certain rights in this material.

briefly discuss problem architecture and its relation to the better understood computer architecture.

## 2 Complex Systems and Space-Time Picture

### 2.1 Problems and Computers

For this article, we shall consider that a *complex system* is a large collection of, in general, disparate members. Those members have, in general, a dynamic connection between them; a dynamic complex system evolves by a statistical or deterministic set of rules which relate the complex system at a later "time" to its state at an earlier "time". Complex systems studied in chemistry and physics, such as a protein or the universe, obey rules that we believe we understand more or less accurately. The military play war games, which is the complex system formed by a military engagement. This and more general complex systems found in society, obey less clear rules. One particular important class of complex systems is that of the *complex computer*. In the case of the hypercube, such as the nCUBE-1,2 or other multicomputers such as the Intel Paragon or Thinking Machines CM-5, the basic entity in the complex systems is a conventional computer and the connection between members is a communication channel implemented either directly in VLSI, on a PC board, or as a set of wires or optical fibers. In another well-known complex computer, the brain, the basic entity is a neuron and an extremely rich interconnection is provided by axons and dendrites.

Mapping one complex system onto another is often important. Solving a problem consists of using one complex system, the *complex processor*, to "solve" another complex system, the *complex problem*. In building a house, the complex processor is a team of masons, electricians, and plumbers, and the complex problem is the house itself. In this article, we are mainly interested in the special case where the complex processor is a complex computer and then modeling or simulating a particular complex problem involves mapping it onto the complex computer. In this case, the map of the complex problem onto the complex computer involves *decomposition*. We can consider the complex problem as an *algorithm* applied to a *data domain*. We divide the data domain into pieces which we call *grains* and place one grain in each node of the concurrent computer.

If we consider a typical matrix algorithm such as multiplication

$$a_{ij} = \sum_k b_{ik}\, c_{kj} \tag{1}$$

we have a data domain formed by the matrix elements, which we generally call *members*. The algorithm (1) defines a graph connecting these members and these connected members form a complex system. The standard *decomposition* involves submatrices stored in each node. Edges of the graph connecting members in different submatrices (i.e., members of the complex system stored in separate nodes of the complex computer) need to be treated especially. To be precise, in the map

| Complex Problem | → | Complex Computer |
|---|---|---|
| Members | map into | memory locations |
| Internal Connections | map into | arithmetic operations |
| Internode or "cut" connections | map into | communication followed by arithmetic operations |

In Section 3, we will be considering topological properties of complex systems which correspond to the map

| Complex Problem | → | Topological Structure |
|---|---|---|
| Members | map into | points in a space geometric |
| Connections | map into | (nearest neighbor) structure |

In the optimal decomposition studies in Section 3 and Section 4, we will be considering dynamic properties of complex systems for which it will be useful to consider the map

| Complex Problem | → | Discrete Physical System |
|---|---|---|
| Members | map into | particles or strings |
| Connections | map into | force between particles or strings |

We see that different classes of complex systems realize their members and interconnection in different ways. We find it very useful to map general systems into a particular class which have a particular choice for members and interconnects. To be precise, complex systems have interconnects that can be geometrical, generated by forces, electrical connection (e.g., wire), structural connection (e.g., road), biological channels or symbolic relationships defined by the laws of arithmetic. We map all these interconnects into electrical communication in the multicomputer implementation. On the other hand, in the simulated annealing approach to load balancing, we map all these interconnects to forces.

## 2.2 Space-Time Picture

The above discussion was essentially static and although this is an important case, the full picture requires consideration of dynamics. We now "define" space and time for a general complex system.

We associate with any complex system a *data domain* or *"space"*. If the system corresponds to a real or simulated physical system, then this data domain is a typically three-dimensional space. In such a simulation, the system consists of a set of objects labelled by index $i$ and is determined by the positions $\underline{x}_i(t)$ at each time $t$. The data domain consists of a set of interconnected nodes and this forms what we call the *computational graph*. This is defined by a time slice of the full complex system.

Other complex systems have more abstract data domains:

1. In a computer chess program, the data domain or "space" is the pruned tree-like structure of possible moves.

2. In matrix problems, the data domain is either a regular two-dimensional grid for full matrices or an irregular subset of this for sparse matrices.

3. In a complex computer defined in Section 2.1, the computational graph of a multicomputer is formed by the individual nodes with the interconnection of the graph determined by the topology (architecture) of the multicomputer. We could enrich this complex system by looking with finer resolution into the computer node itself which can be considered as a set of connected components—chips or transistors depending on detail required.

In a physical simulation, the complex system evolves with time and is specified by the nature of the computational graph at each time. If we are considering a statistical physics or Monte Carlo approach, then we no longer have a natural time associated with the simulation. Rather, the complex system is evolved iteratively or by Monte Carlo sweeps. We will find it useful to view this evolution or iteration label similarly to time in a simple time stepped simulation. We thus consider a general complex system defined by a data domain, which is a structure given by its computational graph. This structure is extended in "time" to give the "space"-"time" cylinders. In our previous examples

1. Chess: time labels depth in tree

2. Matrix Algebra: time labels iteration count in iterative algorithms or "eliminated row" in a traditional full matrix algorithm such as Gaussian elimination.

3. The time dependence of a complex computer is just the evolution given by executed instructions. SIMD machines give an essentially static or synchronous time dependence, whereas MIMD machines can be very dynamic. We will later discuss in Section 5, an interesting class of problems and a corresponding way of using MIMD machines, called loosely synchronous. These are microscopically dynamic or temporally irregular but become synchronous when averaged over macroscopic time intervals.

We expand the discussion of temporal properties in Section 4.

***FIGURE 1***

Figure 1: Computation and Simulation as a Series of Maps

In many areas, one is concerned with mapping one complex system into another. For instance, simulation or modeling consists of a map

$$\text{Nature (or system to be modelled)} \underset{\text{theory}}{\overset{\text{map}}{\longrightarrow}} \text{Idealization or Model} \qquad (2)$$

This map would often be followed by a computer simulation that can be broken up into several maps shown in Figure 1.

Nature, the model, the numerical formulation, the software, and the computer are all complex systems. Typically, one is interested in constructing the maps to satisfy certain goals, such as agreement of model with effects seen in nature or running the computer simulation in a minimum time. In these cases, one gets a class of optimization problems associated with the complex systems. One approach is the use of simulated annealing or neural network methods to address these optimization problems. These are methods to minimize the energy function, which is associated with the general physical system given by the space-time analogy. The energy function is the analytic form that expresses the goal described above. Typically, in studying performance, the energy function would be the execution time of the problem on a computer. For software engineering, the energy function would also reflect user productivity.

Another interesting issue is the loss of information implicit in the successive maps of Figure 1. As reviewed in Section 5, we can discuss key problems in the design of software systems in terms of minimizing information loss.

We believe that the structure of all the complex systems in Figure 1 is interesting. They can be quite different. Consider, for instance, a computational fluid dynamics study of airflow where Figure 1 becomes

$$\begin{aligned} &S_0(\text{flow around airframe}) \rightarrow S_1(\text{molecular picture}) \rightarrow S_2(\text{continuum}) \rightarrow \\ &S_3(\text{numerical method}) \rightarrow S_4(\text{virtual problem}) \rightarrow S_5(\text{final computer}) \quad (3) \end{aligned}$$

$S_0$ is nature.

$S_1$ is a (finite) collection of molecules interacting with long range Van der Waals and other forces. This interaction defines a complete interconnect between all members of the complex system $S_1$.

$S_2$ is the infinite degree of freedom continuum with the fundamental entities as infinitesimal volumes of air connected locally by the partial differential operator of the Navier Stokes equation.

$S_3 = S_{\text{num}}$ could depend on the particular numerical formulation used. Multigrid, conjugate gradient, direct matrix inversion and alternating gradient would have very different structures in the direct numerical solution of the Navier Stokes equations. The more radical cellular automata approach would be quite different again.

$S_4 = S_{\text{HLSoft}}$ would depend on the final computer being used and division between high and low level in software. The label HLSoft denotes "High Level Software".

$S_5 = S_{\text{comp}}$ would be $S_{\text{HLSoft}}$ embroidered by the details of the hardware communication (circuit or packet switching, wormhole or other routing). Further, we would often need to look at this complex system in greater resolution and expose the details of the processor node architecture.

## 3 Spatial Properties of Complex Problems and Complex Computers

### 3.1 System Size

The *size* $N$ of the complex system is an obviously important property. Note that we think of a complex system as a set of *members* with their spatial structure evolving with time. Sometimes, the time domain has a definite "size" but often one can evolve the system indefinitely in time. However, most complex systems have a natural spatial size with the spatial domain consisting of $N$ members. In the matrix example, Gaussian elimination had $n^2$ spatial members evolving for a fixed number of $n$ "time" steps. As usual, the value of spatial size $N$ will depend on the granularity or detail with which one looks at the complex system. One could consider a parallel computer at the level of transistors with very large value of $N$, but usually we look at the processor node as the fundamental entity and define the spatial size of a parallel computer viewed as a complex system, by the number $N_{\text{proc}}$ of processing nodes.

Consider mapping a finite difference simulation with $N_{\text{num}}$ grid points onto a parallel machine with $N_{\text{proc}}$ processors. An important parameter is the *grain size n* of the resultant decomposition. We can introduce the problem *grain size* $n_{\text{num}} = N_{\text{num}}/N_{\text{proc}}$ and the computer *grain size* $n_{\text{mem}}$ as the memory contained in each node of the parallel computer. Clearly we must have,

$$n_{\text{num}} < n_{\text{mem}} \tag{4}$$

if we measure memory size in units of seismic grid points. More interestingly, we will later in Equation 5 relate the performance of the parallel implementation of the seismic simulation to $n_{\text{num}}$ and other problems and computer characteristics. We find that in many cases, the parallel performance only depends on $N_{\text{num}}$ and $N_{\text{proc}}$ in the combination $N_{\text{num}}/N_{\text{proc}}$ and so *grain size* is a critical parameter in determining the effectiveness of parallel computers for a particular application.

## 3.2 Performance Model for a Multicomputer

The next set of parameters describe the topology or structure of the spatial domain associated with the complex system. The simplest parameter of this type is the geometric dimension $d^{\text{geom}}$ of the space. Our early parallel computing used the binary hypercube of dimension $d$, which has $d^{\text{geom}} = d$ as its geometric dimension. This was an effective architecture because it was richer than the topologies of most problems. Thus, consider mapping a problem of dimension $d_{\text{num}}$ onto a computer of dimension $d_{\text{comp}}$. Suppose the software system preserves the spatial structure of the problem and that $d_{\text{HLSoft}} = d_{\text{num}}$. Then, one can show that the parallel computing overhead $f$ has a term due to internode communication that has the form,

$$f_C \propto \frac{N^{\alpha}_{\text{proc}}}{n^{\beta}_{\text{num}}} \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{5}$$

with parallel speedup $S$ given by

$$\begin{aligned} S &= \frac{N}{1+f_C} \\ \text{or } f_C &= \frac{N}{S} - 1 \\ &= \left(\frac{1}{\text{efficiency } \epsilon}\right) - 1 \end{aligned} \tag{6}$$

The communication overhead $f_C$ depends on the problem *grain size* $n_{\text{num}}$ and computer complex system $N_{\text{proc}}$. It also involves two parameters specifying the parallel hardware performance. These are:

- $t_{\text{calc}}$: The typical time required to perform a generic calculation. For scientific problems, this can be taken as a floating point calculation

$$\begin{aligned} a &= b * c \\ \text{or } a &= b + c \end{aligned}$$

- $t_{\text{comm}}$: The typical time taken to communicate a single word between two nodes connected in the hardware topology.

The definitions of $t_{\mathrm{comm}}$ and $t_{\mathrm{calc}}$ are imprecise above. In particular, $t_{\mathrm{calc}}$ depends on the nature of node and can take on very different values depending on the details of the implementation; floating point operations are much faster from registers than from slower parts of the memory hierarchy. On systems built from processors like the Intel i860 chip, these effects can be large; $t_{\mathrm{calc}}$ could be $.0125\mu$ sec from registers (80 megaflop) and a factor of ten larger when the variables $a, b$ are fetched from dynamic RAM. Again, communication speed $t_{\mathrm{comm}}$ depends on internode message size (a software characteristic) and the latency (startup time) and bandwidth of the computer communication subsystem.

Returning to Equation 5, we really only need to understand here that the term $t_{\mathrm{comm}}/t_{\mathrm{calc}}$ indicates that communication overhead depends on relative performance of the internode communication system and node (floating point) processing unit. A real study of parallel computer performance would require a deeper discussion of the exact values of $t_{\mathrm{comm}}$ and $t_{\mathrm{calc}}$. More interesting here is the dependence $(N^{\alpha}_{\mathrm{proc}}/n^{\beta}_{\mathrm{num}})$ on the number of processors $N_{\mathrm{proc}}$ and problem *grain size* $n_{\mathrm{num}}$. As described above, *grain size* $n_{\mathrm{num}} = N_{\mathrm{num}}/N_{\mathrm{proc}}$ depends on both the problem and the computer. The values of $\alpha$ and $\beta$ are given by

$$\beta = \frac{1}{d_{\mathrm{num}}} \tag{7}$$

independent of computer parameters while if

$$\begin{aligned} d_{\mathrm{num}} < d_{\mathrm{comp}} \quad &, \quad \alpha = 0 \\ \text{and if } d_{\mathrm{num}} > d_{\mathrm{comp}} \quad &, \quad \alpha = \left(\frac{1}{d_{\mathrm{comp}}} - \frac{1}{d_{\mathrm{num}}}\right) \end{aligned} \tag{8}$$

The results in Equation 8 quantify the penalty, in terms of a value of $f_C$ that increases with $N_{\mathrm{proc}}$, for a computer architecture that is less rich than the problem architecture. An attractive feature of the hypercube architecture is that $d_{\mathrm{comp}}$ is large and one is essentially always in the regime governed by $\alpha = 0$ in Equation 8. Recently, there has been a trend away from rich topologies like the hypercube towards the view that the node interconnect should be considered as a routing network or switch to be implemented in the very best technology. The original MIMD machines from Intel, nCUBE and Ametek all used hypercube topologies as did the SIMD Connection Machine CM-1, CM-2. The nCUBE-2 introduced in 1990, still uses a hypercube topology but both it and the second generation Intel iPSC/2 used more sophisticated routing. The latest Intel Paragon and Touchstone Delta and Symult (ex Ametek) 2010 use a two-dimensional mesh with wormhole routing. It is not clear how to incorporate these new node interconnects into the above picture, and further research is needed here. Presumably, we would need to add new complex system properties and perhaps generalize the definition of dimension $d_{\mathrm{comp}}$ as we will see below is in fact necessary for Equation 5 to be valid for problems whose structure is not geometrically based.

***FIGURE 2 ***

Figure 2: The Information Density and Flow in a General Complex Systems with Length Scale $L$

## 3.3 System Dimension

Returning to equations 5, 6, 7, and 8 we note that we have not properly defined the correct dimension $d_{\text{num}}$ or $d_{\text{comp}}$ to use. We have implicitly equated this to the natural *geometric dimension* but this is not always correct. This is illustrated by the complex system $S_{\text{num}}$ consisting of a set of particles in three dimensions interacting with a long range force such as gravity or electrostatic charge. The geometric structure is local with $d_{\text{num}}^{\text{geom}} = 3$ but the complex system structure is quite different; all particles are connected to all others. As described in Chapter 3 of [3], this implies that $d_{\text{num}}^{\text{system}} = 1$ whatever the underlying geometric structure. We define the *system dimension* $d^{\text{system}}$ for a general complex system to reflect the system connectivity. Consider Figure 2 which shows a general domain $D$ in a complex system. We define the *volume* $V_D$ of this domain by the information in it. Mathematically, $V_D$ is the computational complexity needed to simulate $D$ in isolation. In a geometric system

$$V_D \propto L^{d^{\text{geom}}} \tag{9}$$

where $L$ is a geometric length scale. The domain $D$ is not in general isolated and is connected to the rest of the complex system. Information $I_D$ flows in $D$ and again in a geometric system. $I_D$ is a surface effect with

$$I_D \propto L^{d^{\text{geom}}-1} \tag{10}$$

If we view the complex system as a graph, $V_D$ is related to the number of links of the graph inside $D$ and $I_D$ to the number of links cut by the surface of $D$. Equation 9 and Equation 10 are altered in cases like the long range force problem where the complex system connectivity is no longer geometric. We define the system dimension to preserve the surface versus volume interpretation of Equation 10 compared to Equation 9. Thus, generally we define

$$I_D \propto V_D^{1-1/d^{\text{system}}} \tag{11}$$

With this definition of *system dimension* $d^{\text{system}}$, we will find that Equation 5, 6, 7, and 8 essentially hold in general. In particular for the long range force problem, one finds $d^{\text{system}} = 1$ independent of $d^{\text{geom}}$.

## 3.4 Physical Analogy

In the previous three subsections, we described static spatial properties of complex systems which were relevant for computation. These included size, topology (geometric dimension) and the information dimension. We will find new ideas when we consider problems that are spatially irregular and perhaps vary slowly with time. A simple example would be a large scale astrophysical simulation where the use of a parallel computer required that the universe be divided into domains that, due to the gravitational interactions will change as the simulation evolves.

Load Balancing can affect crucially the performance of a computation executing on a parallel machine. By "load balance" we refer to the amount of cpu idling occurring in the processors of the concurrent computer: a computation for which all processors are continually busy (and doing useful-non-overlapping work) is considered perfectly balanced. This balance is often not trivial to achieve, however. The problem of distributing a computation in an efficient manner into a parallel machine can be fruitfully attacked via simulated annealing and other physical optimization methods [6]–[12].

As described in the previous section, a key to parallel computing is to split the underlying spatial domain into grains which each correspond to a process as far as the operating system is concerned. We will take a naive software model where there is one process associated with each of the fundamental members of the simulated system, i.e., with each "particle" in the astrophysical simulation. This is not practical with current software systems as it gives high context switching and other overheads. However, it captures the essential issues.

The processes will need to communicate with one another in order for the computation to proceed. Assume that the processes and their communication requirements are changing with time—processes can be created or destroyed, communication patterns will move. This is the natural choice when one is considering timesharing the parallel computer, but can also occur within a single computation. It is the task of the operating system to manage this set of processes, moving them around if necessary, so that the parallel computer is used in an efficient manner.

The operating system performs two primary tasks. First, it must monitor the ongoing computation so as to detect bottlenecks, idling processors and so on. Secondly, it must modify the distribution of processes and also the routing of their associated communication links so as to improve the situation. In general, it is very difficult to find the optimum way of doing this—in fact, this is an NP complete problem. Approximate solutions, however, will serve just as well. We will be happy if we can realize a reasonable fraction (let's say 80%) of the potential computing power of the parallel machine for a wide variety of computations. We will see in what follows that the operating system functions as a "heat bath", keeping the computation "cool" and therefore near its "ground state" (optimal solution).

One can usefully think of a parallel computation in terms of a physical analogy. Treat the processes as "particles" free to move about in the "space" of the parallel machine. Minimizing the total execution time of the parallel computation, formally requires that one minimize:

$$\max_{\text{nodes } i} C_i \tag{12}$$

where $C_i$ is the total computation time for calculation and communication. We choose to replace this mini-max problem by a least squares [9] minimization of

$$E = \sum_i C_i^2 \tag{13}$$

Suppose $m(m')$ label the nodal points of the computational graph. Then

$$C_i = \sum_{m \epsilon i} \left[ \sum_{\substack{m' \\ \text{linked} \\ \text{to } m}} Comm(m, m') + Calc(m) \right] \tag{14}$$

where it takes time $Calc(m)$ to simulate $m$ and time $Comm(m', m)$ to communicate necessary information from $m'$ to $m$. If we consider the case where we can neglect the quadratic communication terms, then

$$\begin{aligned} C_i^2 \;&=\; \text{const.} \sum_{\substack{m,m' \\ \text{for } m \text{ in } i \\ \text{and } m' \text{ linked} \\ \text{to } m}} \text{Comm}\,(m, m') \\ &+ \sum_{\substack{\text{with} \\ m,m' \\ \text{in } i}} \text{Calc}\,(m)\ \text{Calc}\,(m') \end{aligned} \tag{15}$$

The last term in the Hamiltonian (Equation 15) corresponds to the requirement of load balancing which acts as a short range, repulsive "force", causing the particles, and thereby the computation, to spread throughout the parallel computer in an evenhanded, balanced manner. The potential is indeed short range where range is measured by distance between nodes in the space of the complex computer. The last term in Equation 15 is zero unless particles $m$ and $m'$ are at the same place, i.e., in same node.

A conflicting requirement to that of load balancing is shown in the first term of Equation 15 as interparticle communications—the various parts of the overall computation need to communicate with one another at various times. If the particles are far apart (distance being defined as the number of communication steps separating them) large delays will occur, slowing down the computation. Thus, this represents a long range, attractive force between those pairs of particles which need to communicate with one another. This force is proportional to the amount of communication traffic between the particles, so that heavily communicating parts of the computation will coalesce and tend to stay near one another in the computer.

We have given in Equation 15, and described qualitatively above, a "Hamiltonian" for parallel computation, which the operating system must try to minimize, and if possible find the ground state. We already noted that exact minimization is not necessary—we have already "wasted" some computational power using convenient high level languages—we can surely afford to lose another 20% to load imbalance, so we can think of the operating system as a heat bath which keeps the computation as cool as possible. Most scientific simulations change slowly with time and redistribution of processes by the operating system can be gradual. Thus, we can think of the computation as being in *adiabatic* equilibrium at a complex system *temperature* $T_{\text{problem}}$ which reflects the ease of finding a reasonable minimum. $T_{\text{problem}}$ will be larger for those problems which change more rapidly and where the operating system does not have "time" to find as good an equilibrium.

The elegant physical analogy makes it plausible that simulated annealing is an appropriate minimization technique for Equation 13. It corresponds to using the normal Monte Carlo method to finding the ground state of the associated physical system.

In [13, 14, 15], we explore this analogy further and find that at low temperatures, the parallel computation exhibits a phase transition controlled by the relative strengths of the terms in Equation 15. This phase transition corresponds to a switch between different styles of decomposition.

## 4 Temporal Properties of Complex Problems and Complex Computers

### 4.1 The String Formalism for Dynamic Problems

In the previous section, we thought of a problem (the complex system $S_{\text{num}}$ or $S_{\text{HLSoft}}$) as a graph (the *computational graph*) with vertices labelled by the system member $m$ and edges corresponding to the linkage between members established by the algorithm. This is a good picture for what we called *adiabatic* problems that change slowly with time. In this case, it makes sense to think of slicing the "space-time" cylinder formed by the complex system and just consider the computational graph—the spatial structure at fixed time. However, this is not appropriate for rapidly varying or *dynamic* problems—those with high temperature $T_{\text{problem}}$ in the language of Section 3.4. For such problems, the operating system cannot "keep up" with the variation of the computational graph—the graph changes significantly over the time period that the operating system takes to partition the computational graph.

In *adiabatic* problems, our physical analogy was that of members mapped in particles interacting by forces given by the member interconnect. One might imagine that a reasonable analogy for *dynamic* problems was to add a kinetic energy term to give time dependence to the member positions. I do not understand how to do this. Rather, we change the analogy to that of members corresponding to strings representing the world lines of members moving in time (measured by the complex computer).

We make this more precise with a *dynamic* complex system whose members are labelled by $m$. At computer time $t$, member $m$ is located at position $\underline{x}_m(t)$. $\underline{x}$ is a position in the complex computer space. At its simplest $\underline{x}$ is just a node number, but we can look at a finer resolution and consider $x$ as a position in the global computer memory. This allows one, in principle at least, to set up a formalism in which one can study the full memory hierarchy of the system including caches and register use. Each member now corresponds not just to a position $\underline{x}_m$ but to a world-line $\{x_m(t)\}$. The execution time $T_{\text{par}}$ on a parallel machine is a functional of the world lines

$$T_{\text{par}} \equiv T_{\text{par}}\left(\{\underline{x}_0(t)\} \ldots\ldots \{\underline{x}_m(t)\} \ldots\right) \qquad (16)$$

The structure of the original dynamic complex system leads to an expression for Equation 16 which is similar to the simpler Equation 15. There is a repulsive force between world lines corresponding to load balancing. There is an attractive force corresponding to the dynamic interconnection between the members $m$. The details of this depend on the relation between clock time $t$ and the simulation time $t_m$ of each member $m$.

The most straightforward approach to minimize $T_{\text{par}}$ would be simulating annealing with the basic "move" being a change

$$\{\underline{x}_m(t)\} \rightarrow \{\underline{x}_m(t)\}'$$

which is typically local in both $\underline{x}$ and $t$. This gives a formalism similar to quantum chemistry or lattice gauge theories. One can also use a neural network formalism which generalizes the original approach of Hopfield and Tank to the Travelling Salesman Problem [16]. These points are described in greater detail in references [8, 10, 15, 17].

We have applied these ideas to message routing in a network [18] and more generally combining networks which implement global reduction formulae such as forming a set of sums

$$y_j = \sum_i M_{ji}\, x_i \qquad (17)$$

where $y_j$, $M_{ji}$, and $x_i$ are all distributed over the nodes of a parallel computer.

A very initial examination was given in [19] of the application of these ideas to register allocation for compilers. We have explored more deeply the application of these methods to multi-vehicle navigation [8, 20]. Now $\{\underline{x}_m(t)\}$ is the path of vehicle $m$ in a two or three dimensional space with $m$ at position $\underline{x}_m$ at time $t$.

## 4.2 Memory Hierarchy

The discussion of Section 3 can be thought of as the mapping of the spatial structure of the problem's complex system into the spatial structure of the computer. The dimensionality relation of Equation 8 explains how these structures must be related to get good performance.

We find a remarkable analogy with the temporal structure of the problem corresponding to the memory hierarchy of the computer. In particular, we see that the well known methods for improving the performance of caches and registers, correspond to blocking (clumping) the problem in its time direction. This is analogous to blocking in space (as quantified by the grain $n_{\mathrm{num}}$ dependence of Equation 5) to improve performance of a parallel machine. In particular, the overhead $f_{\mathrm{H}}$ due to cache misses, i.e., to reading and writing between cache and main memory, takes exactly the same form as Equation 15. To obtain $f_{\mathrm{H}}$, one must substitute $t_{\mathrm{mem}}$ (a typical time to read a word to cache) for $t_{\mathrm{comm}}$ and $n_{\mathrm{time}}$ for $n_{\mathrm{num}}$. $n_{\mathrm{time}}$ is the temporal blocking factor—the number of iterations in the problem between cache flushes.

High performance computer architectures exploit data locality with a memory hierarchy implemented either as a multilevel cache and/or with distributed memory on a parallel machine. Good use of cache requires blocking in time; good use of distributed memory requires blocking in space. In general, full space-time blocking is required.

## 5 Problem Architectures and Parallel Software

In a series of papers, we have developed a qualitative theory of the architectures of problems [1, 21, 22, 23]. This is summarized in Table 1, which introduces five general classes of problem classes. This is analogous to the well-known classification of parallel computer architectures into SIMD and MIMD. We now return to the concept of Figure 1—namely, computation is map between problem and computer, and software is an expression of this map. We have explored in depth this concept of problem architecture and its use for clarifying which problems run well on SIMD machines and which on MIMD. One can also understand which problem classes parallelize naturally on massively parallel machines. Here, we just describe the consequences for software, which are summarized in Table 1.

We believe that successful software models will be built around problem and not machine architecture. We see that some of the current languages—both old and new—are flawed because they do not use this principle in their design. The language often reflects artifacts of a particular machine architecture and this naturally leads to nonportable codes that can only be run on the machine whose architecture is expressed by the language. On the other hand, if the language expresses properly the problem structure, then a good compiler should be able to map into a range of computer architectures.

Table 1: Architectures for Five Problem Classifications

- **Synchronous: Data Parallel**

  Tightly coupled. Software needs to exploit features of problem structure to get good performance. Comparatively easy, as different data elements are essentially identical. *Candidate software paradigms: High Performance Fortran, Parallel Fortran 77D, Fortran 90D, CMFortran, Crystal, APL, C++.*

- **Loosely Synchronous: Data Parallel**

  As above but data elements are not identical. Still parallelizes due to macroscopic time synchronization. *Candidate software paradigms: may be extensions of the above. C (Fortran) message passing is currently only guaranteed method!*

- **Asynchronous**

  Functional (or data) parallelism that is irregular in space and time. Often loosely coupled and so need not worry about optimal decompositions to minimize communication. Hard to parallelize (massively). *Candidate software paradigms: PCN, Linda, object-oriented approaches.*

- **Embarrassingly Parallel**

  Independent execution of disconnected components. *Candidate software paradigms: Several approaches work? PCN, Linda, Network Express, ISIS.*

- **A=LS (Loosely Synchronous Complex)**

  Asynchronous collection of loosely synchronous components where these program modules can be parallelized. *Candidate software paradigms: PCN, Linda, ADA, controlling modules written in synchronous or loosely synchronous fashion.*

We can illustrate this with Fortran 77, which we can view as embodying the architecture of a sequential machine. Thus, software written in Fortran 77 maps the space-time structure of the original complex system into a purely temporal or control structure. The spatial (data) parallelism of the problem becomes purely temporal in the software, which implements this as a DO loop. Somewhat perversely, a parallelizing compiler tries to convert the temporal structure of a DO loop back into spatial structure to allow concurrent execution on a spatial array of computers. Often parallelizing compilers produce poor results as the original map of the problem into sequential Fortran 77 has "thrown away" information necessary to reverse this map and recover unambiguously the spatial structure. The first (and some ongoing) efforts in parallelizing compilers tried to directly "parallelize the DO loops". This seems doomed to failure in general as it does not recognize that in nearly all cases the parallelism comes from spatial and not control (time) structure. Thus, we are working with Kennedy at Rice and others on a parallelizing compiler FortranD where the user adds additional information to tell the compiler about the spatial structure. We are optimistic that the resultant Fortran D project [22]–[26] will be successful for the synchronous and loosely synchronous problem classes defined in Table 1.

Most languages do not express and preserve space time structure. Array languages such as APL and Fortran 90 are examples of data parallel languages that at least partially preserve the space time structure of the problem in the language. Appropriate class libraries can also be used in C++ to achieve this goal. We expect that development of languages which better express problem structure will be essential to get good performance with an attractive user environment on large scale parallel computers. The results in Section 4.2 show that data locality is critical in sequential high performance (hierarchical memory) machines as well. Thus, we would expect that the use of languages that properly preserve problem structure will lead to better performance on all computers.

**Acknowledgements**

This work has been collaborative with many people, especially W. Furmanski and S. Otto and I would like to thank them and my colleagues in the Caltech Concurrent Computation Program for their insight and help.

## References

[1] Angus, I. G., Fox, G. C., Kim, J. S., and Walker, D. W. *Solving Problems on Concurrent Processors: Software for Concurrent Processors*, volume 2. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1990.

[2] Fox, G. C. "Questions and unexpected answers in concurrent computation," in J. J. Dongarra, editor, *Experimental Parallel Computing Archi-*

*tectures*, pages 97–121. Elsevier Science Publishers B.V., North-Holland, 1987. Caltech Report C3P-288.

[3] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1988.

[4] Fox, G. C. "The hypercube and the Caltech Concurrent Computation Program: A microcosm of parallel computing," in B. J. Alder, editor, *Special Purpose Computers*, pages 1–40. Academic Press, Inc., 1988. Caltech Report C3P-422.

[5] Fox, G. C., Messina, P. C., and Williams, R. D., editors. *Parallel Computing Works*. Morgan Kaufmann Publishers, 1992. In preparation.

[6] Flower, J., Otto, S., and Salama, M. "Optimal mapping of irregular finite element domains to parallel processors." Technical Report C3P-292b, California Institute of Technology, August 1987. In Proceedings, Symposium on Parallel Computations and Their Impact on Mechanics, ASME Winter Meeting, Dec. 14–16, Boston, Mass.

[7] Fox, G. C., and Furmanski, W. "Load balancing loosely synchronous problems with a neural network," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, pages 241–278. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-363b.

[8] Fox, G. C., Furmanski, W., Ho, A., Koller, J., Simic, P., and Wong, Y. F. "Neural networks and dynamic complex systems." Technical Report C3P-695, California Institute of Technology, December 1988. Proceedings of 1989 SCS Eastern Conference , Tampa, Florida, March 28–31, 1989.

[9] Fox, G. C. "A review of automatic load balancing and decomposition methods for the hypercube," in M. Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 63–76. Springer-Verlag, 1988. Caltech Report C3P-385b.

[10] Fox, G. C. "Physical computation," *Concurrency: Practice and Experience*, 3(6):627–653, December 1991. Special Issue: Practical Parallel Computing: Status and Prospects. Guest Editors: Paul Messina and Almerico Murli. SCCS-2b, C3P-928b, CRPC-TR90090.

[11] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. "Optimization by simulated annealing," *Science*, 220(4598):671–680, May 1983.

[12] Williams, R. D. "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency: Practice and Experience*, 3(5):457–481, October 1991. Caltech Report C3P-913b.

[13] Fox, G., Otto, S. W., and Umland, E. A. "Monte Carlo physics on a concurrent processor," *Journal of Statistical Physics*, 43(5/6), June 1986. Proceedings of the Conference on Frontiers of Quantum Monte Carlo, September 3–6, 1985 at Los Alamos. Caltech Report C3P-214.

[14] Fox, G., and Otto, S. "Concurrent computation and the theory of complex systems," in M. T. Heath, editor, *Hypercube Multiprocessors*, pages 244–268. SIAM, Philadelphia, 1986. Caltech Report C3P-255.

[15] Fox, G. C. "The use of physics concepts in computation," in B. A. Huberman, editor, *Computation: The Micro and the Macro View*, chapter 3. World Scientific Publishing Co. Pte. Ltd., 1992. SCCS-237, CRPC-TR92198. Caltech Report C3P-974.

[16] Hopfield, J. J., and Tank, D. W. "Computing with neural circuits: a model," *Science*, 233:625, 1986.

[17] Fox, G. C. "Approaches to physical optimization." Technical Report C3P-959, California Institute of Technology, April 1991. CRPC-TR91124, SCCS-92, Submitted to SIAM J. Sci. Stat. Comp. for publication.

[18] Fox, G. C., and Furmanski, W. "The physical structure of concurrent problems and concurrent computers," in R. J. Elliott and C. A. R. Hoare, editors, *Scientific Applications of Multiprocessors*, pages 55–88. Prentice Hall, 1988. Caltech Report C3P-493.

[19] Fox, G. C., and Koller, J. G. "Code generation by a generalized neural network: General principles and elementary examples," *Journal of Parallel and Distributed Computing*, 6(2):388–410, 1989. Caltech Report C3P-650b.

[20] Fox, G. C. "Applications of the generalized elastic net to navigation." Technical Report C3P-930, California Institute of Technology, June 1990. Unpublished.

[21] Fox, G. C. "Parallel problem architectures and their implications for portable parallel software systems." Technical Report C3P-967, Northeast Parallel Architectures Center, May 1991. CRPC-TR91120, SCCS-78, Presentation at DARPA Workshop, Providence, Rhode Island, February 28, 1991.

[22] Fox, G. C. "FortranD as a portable software system for parallel computers." Technical Report SCCS-91, Syracuse University, June 1991. Published in the Proceedings of Supercomputing USA/Pacific 91, held in Santa Clara, California. CRPC–TR91128.

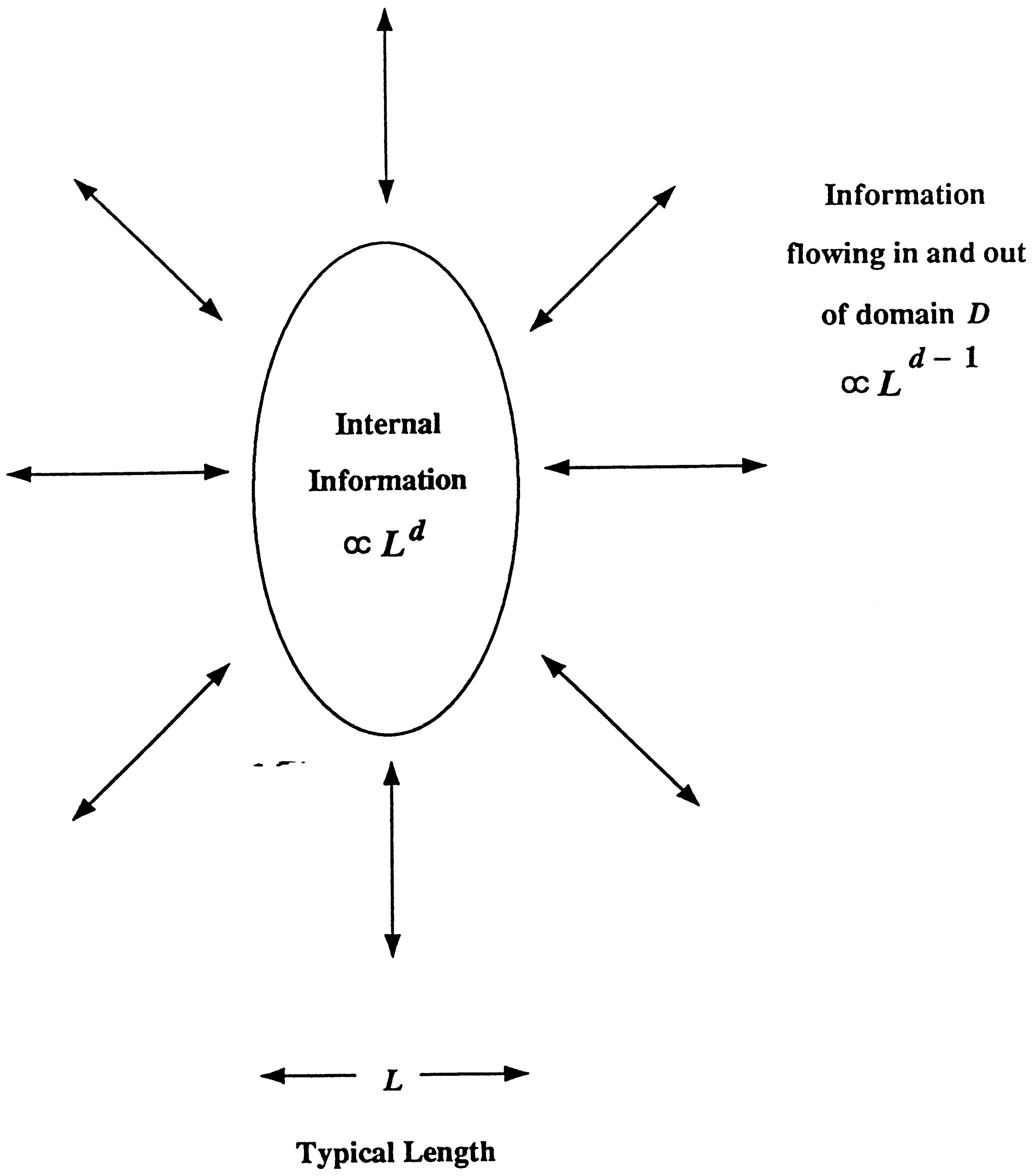
[23] Fox, G. C. "The architecture of problems and portable parallel software systems." Technical Report SCCS-134, Syracuse University, July 1991. Revised SCCS-78b.

[24] Fox, G. C. "Hardware and software architectures for irregular problem architectures," in P. Mehrotra, J. Saltz, and R. Voigt, editors, *Unstructured Scientific Computation on Scalable Microprocessors*, pages 125–160, Cambridge, Massachusetts, 1992. The MIT Press. Scientific and Engineering Computation Series. Held by ICASE in Nags Head, North Carolina. SCCS-111; CRPC-TR91164.

[25] Fox, G. C., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W., and Wu, M.-Y. "Fortran D language specification." Technical Report SCCS-42c, Syracuse University, April 1991. Rice Center for Research in Parallel Computation; CRPC-TR90079.

[26] Wu, M., and Fox, Geoffrey, C. "Fortran 90D compiler for distributed memory MIMD parallel computers." Technical Report SCCS-88b, Syracuse Center for Computational Science, September 1991. CRPC-TR91126.

$$\begin{tikzcd}
\text{Nature} \arrow[r, "\text{Brilliant}", "\text{Idea}"'] & \text{Theory} \arrow[r] & \text{Model} \arrow[dll] \\
\text{Numerical Method} \arrow[dr, "\text{High Level Software}"] & & \\
& \text{Virtual Computer or Virtual Problem} \arrow[dr, "\text{Low Level Software}"] & \\
& & \text{Real Computer}
\end{tikzcd}$$


Information flowing in and out of domain $D$ $\propto L^{d-1}$
Internal Information $\propto L^{d}$
$L$
Typical Length